Spring 5-1-1990

Local Attributes for OAG-Based Attribute Evaluators ; CU-CS-472-90

Masayoshi Ishikawa
University of Colorado Boulder

Revision: 1.4

Local Attributes for OAG-based Attribute Evaluators

Masayoshi Ishikawa

CU-CS-472-90 May, 1990

ABSTRACT: Attribute grammars permit us to give meaning to a string in a context-free language. This meaning is expressed using attributes attached to symbols (terminals and non-terminals) of the underlying context-free grammar. In this paper we extend the concept of attributes, and allow to attach attributes to each production rule too. These rule-attached attributes are often called *local attributes* because of their scoping restriction. They can be defined and referenced only by attribution rules of the production rule to which they are attached. These attributes provide us the best place to store temporary values which can be used later to define other attributes in the same rule. First part of the paper describes the theoretical background to incorporate *local attributes* with OAG-based attribute evaluator generators such as [2,3]. Then it introduces the formal definition of *local attributes* along with a typical example where they are useful.

1. Introduction

Since Knuth [1] proposed attribute grammars(AGs) to describe meanings of context-free languages(CFLs), many compiler generators based on AGs have been implemented [2,3,4,5]. Ordered attributed grammars(OAGs) are a relatively large subclass of AGs originated by [6]. It is known that only polynomial time is required to solve the membership problem(given an AG, test it if it belongs to OAGs or not). [2,7] reported that the class of OAG was powerful enough to define the semantics of commonly used programming languages such as Pascal, C, and Ada.

Currently there are two OAG-based attribute evaluator generating systems in use[2,3]. These systems take an AG given by the user, test it if it is an OAG, and if so it generates an evaluator(compiler) that computes all the values of attributes on the derivation tree at runtime by traversing the nodes of the tree for any legal user input(source program). When an AG is submitted to the system, the order of computations of all the attributes are decided statically by the attribute evaluator generating systems only based on the underlying context-free grammar(CFG) and the dependencies between attributes. The dependencies are relationship between attributes. If one attribute is used to define another, the latter is said to be dependent on the former. If two attributes are dependent each other, the system can not generate an attribute evaluator(compiler) because there is no way to decide which attribute should be computed before the other. These dependencies are often called cyclic dependencies.

Since OAG is a subset of all well defined AGs(AGs that compute all values of attributes effectively for any legal input), it sometimes makes mistakes and recognize acyclic dependencies as cyclic ones. Normalization of dependencies among attributes[8] is a technique that modify the original set of dependencies and decrease the degree of mistakes. However such modifications are not always possible.

This paper describes a situation in which such modification does not work, and present a simple algorithm that allows the normalization always. Then the concepts of *local attributes*, attributes associated with productions of AG, are introduced as a practical application of this algorithm.

2. Preliminaries

We introduce some definitions and notations of AGs and OAGs. We assume the readers are familiar with the notion of AGs, OAGs, and visit-oriented attribute evaluators[6,9].

An attribute grammar AG is a 4-tuple (G, A, R, B) of a CFG G, attributes A, attribution rules R, and conditions B.

The CFG G is a 4-tuple (N, T, P, Z) of nonterminals N, terminals T, productions P, and a unique starting nonterminal symbol Z.

An attribute a of a symbol X is denoted by $X.a$. The set of all attributes associated with X is denoted by $A(X)$.

A production is of the form $p : X_0 \rightarrow X_1 X_2 \cdots X_{np}$, where $X_0 \in N$ and $X_i (1 \leq i \leq np) \in T \cup N$.

An attribution rule of $p : X_0 \rightarrow X_1 X_2 \cdots X_{np}$ is of the form $X_i.a := f(\cdots X_j.b \cdots)$, where $0 \leq i, j \leq np$. The set $R(p)$ consists of all attribution rules of p.

An attribute $X.a$ is said to be defined in the context of p if $X.a := f(\cdots) \in R(p)$. The set of all attributes defined in the context of p is denoted by $AF(p)$.

If an attribute $X.a$ is defined by an attribution rule $X.a := f(\cdots Y.b \cdots)$, we say that $X.a$ is dependent on $Y.b$, or there is a dependency $(Y.b, X.a)$ between $X.a$, and $Y.b$. The set $DDP(p)$ consists of all dependencies specified by some attribution rule in $R(p)$.

The normalized transitive closure of $DDP(p)$, denoted by $NDDP(p)$, is a set given by $NDDP(p) = DDP(p)^+ - \{(X.a, Y.b) | X.a, Y.b \in AF(p)\}$, where $^+$ is a non-reflexive transitive closure operator.

A visit sequence of $p : X_0 \rightarrow X_1 X_2 \cdots X_{np}$ is a sequence of the form $\langle s_1, s_2, \ldots, s_{mp} \rangle$, and is denoted by $vs(p)$. In the visit-oriented attribute evaluator, each visit sequence $vs(p)$ acts as a co-routine when executed that evaluates some values of attributes defined in the context of p, and invokes other visit sequences to make them compute some attributes defined in their contexts. Hence each $s_i (1 \leq i \leq mp)$ is either a definition of an attribute $X.a$ denoted simply by $X.a$, or wth visit to some other visit sequence corresponding to a symbol $X_u (0 \leq u \leq np)$ denoted by $\downarrow_w X_u$. Note that the wth visit to X_0 is denoted by $\uparrow_w X_0$ in [10], but we use $\downarrow_w X_0$ instead.

The complete construction of the visit sequences is beyond the scope of this paper[6,11], so we only list the steps and the necessary conditions for successful construction.

1. For each $p : X_0 \rightarrow X_1 \cdots X_{np} \in P$, we construct $IDP(p)$ by the following algorithm.
   1-1. First let $IDP(p) = DDP(p)$
   1-2. Let $IDP(p) = IDP(p)^+$
   1-3. Let $IDP(p) = IDP(p) \cup \{(X_i.a, X_i.b) | \exists_{q:Y_0 \rightarrow Y_1 \cdots Y_{nq}} X_i = Y_j \wedge (Y_j.a, Y_j.b) \in IDP(q)\}$
   1-4. Repeat step 1-2 and 1-3 until no $IDP(p)$ changes.
2. Condition: each $IDP(p)$ must be acyclic.
3. For each $X \in T \cup N$, construct $IDS(X) = \{(X.a, X.b) | \exists_{p:X_0 \rightarrow X_1 \cdots X_{np}} X = X_i (0 \leq i \leq np) \wedge (X_i.a, X_i.b) \in IDP(p)\}$

4. For each $X \in T \cup N$, construct a partition $A_1(X), A_2(X), \ldots, A_{mx}(X)$ of attributes $A(X)$ from the dependencies in $IDS(X)$ such that $(X.a, X.b) \in IDS(X)$ implies $X.a \in A_i(X) \wedge X.b \in A_j(X) \wedge i \leq j$.
5. For each $p : X_0 \rightarrow X_1 \cdots X_{np}$, construct a visit sequence $vs(p)$ from partitions of $A(X_i)(0 \leq i \leq np)$, and dependencies in $IDP(p)$.

3. Example

Consider the attribute grammar shown in Fig. 1. The dependencies among attributes of the symbol X are not cyclic. However, while constructing IDPs, they are recognized as cycles because of $(X.b, X.c)$ in $DDP(p_1)$ and $(X.c, X.b)$ in $DDP(p_2)$. Consequently we fail to construct visit sequences.

Fig. 1

$$
\begin{aligned}
&T = \{\tau\} \\
&N = \{Z,\ X,\ Y\} \\
\\
&A(Z) = \phi \\
&A(X) = \{X.a,\ X.b,\ X.c\} \\
&A(Y) = \{Y.a\} \\
\\
&p_1 : Z \rightarrow X\ Y \\
&R(p_1) = \{X.a := \text{gen\_label}(),\ X.b := f_1(X.a),\ X.c := f_2(X.b),\ Y.a := 1\} \\
&DDP(p_1) = \{(X.a,\ X.b),\ (X.b,\ X.c)\} \\
\\
&p_2 : Y \rightarrow X \\
&R(p_2) = \{X.a := \text{gen\_label}(),\ X.b := f_4(X.c),\ X.c := f_3(X.a, Y.a)\} \\
&DDP(p_2) = \{(X.c,\ X.b),\ (X.a,\ X.c),\ (Y.a,\ X.c)\} \\
\\
&p_3 : X \rightarrow \tau \\
&R(p_3) = \phi \\
&DDP(p_3) = \phi
\end{aligned}
$$

Fig. 2

$$
\begin{aligned}
&NDDP(p_1) = \phi \\
&NDDP(p_2) = \{(Y.a,\ X.b),\ (Y.a,\ X.c)\} \\
&NDDP(p_3) = \phi
\end{aligned}
$$

In [11], they suggested to use Bochman normal form of dependencies [8] in order to avoid this type of bogus cycles. This requires a slight modification of the construction steps mentioned

earlier. In step 1-1, now we let $IDP(p) = NDDP(p)$ in place of $IDP(p) = DDP(p)$. The normalized dependencies of the same AG are shown in Fig. 2, and the visit sequences listed in Fig. 3 are successfully derived from the modified construction steps. The declarative nature of AGs should guarantee that re-evaluations of attribution rules yield same values. Although this theory is the basis of the normalization of dependencies, we sometimes find it unacceptable.

In this case, we can not directly use the attribution rules in Fig. 1 to evaluate the attributes. For instance at production p_2 in Fig. 1, the attribute $X.b$ is defined by $X.b := f_4(X.c)$. But the visit sequence $vs(p_2)$ in Fig. 3 tell us to evaluate $X.b$ before $X.c$. Here the declarative nature of AG is supposed to enable us to rewrite the original attribution rules in Fig. 1 as shown in Fig. 4 to compute the attribute values in accordance with the derived visit sequences. However this is not true in practice sometimes. Suppose gen_label() is a function to generate a unique number every time invoked so that it can be used for a label in the code generation phase of a compiler.

Fig. 3

$$
\begin{aligned}
vs(p_1) &= \langle\ X.b,\ X.c,\ X.a,\ \downarrow X,\ Y.a,\ \downarrow Y,\ \downarrow Z\ \rangle \\
vs(p_2) &= \langle\ X.b,\ X.c,\ X.a,\ \downarrow X,\ \downarrow Y\ \rangle \\
vs(p_3) &= \langle\ \downarrow X\ \rangle
\end{aligned}
$$

Unfortunately it is obvious that the visit sequences in Fig. 3 and the new evaluation scheme in Fig. 4 do not give us the same result as the original ones(Fig. 1). For example, consider the value of $X.c$ in p_2. The re-evaluation of gen_label() produces a different result from what $X.a$ gets assigned in the rule $X.a :=$ gen_label(). In the original attribution rule $X.c := f_3(X.a, Y.a)$, $X.c$ should depend on the same value as $X.a$ holds, but not the new value returned by gen_label() . This is not the intention of the person who originally specified the attribution rules.

Fig. 4

$$
\begin{aligned}
R(p_1) &= \{X.a := \text{gen\_label}(),\ X.b := f_1(\text{gen\_label}()),\ X.c := f_2(f_1(\text{gen\_label}())),\ Y.a := 1\} \\
R(p_2) &= \{X.a := \text{gen\_label}(),\ X.b := f_4(f_3(\text{gen\_label}(), Y.a)),\ X.c := f_3(\text{gen\_label}(), Y.a)\} \\
R(p_3) &= \phi
\end{aligned}
$$

Simple modification of $vs(p_2)$ helps this situation. Looking at the visit sequence $vs(p_2)$ in Fig. 3, we realize that $X.c$ appears before $X.a$ and even worse $X.a$ appears before both $X.c$ and $X.a$. The problems come from the fact that this order does not reflects the original dependencies in Fig. 1. If they had been located in the order of $X.a$, $X.c$, and $X.b$ in $vs(p_2)$, there would not have been any problems; we could avoid the bogus cycles and successfully evaluate the attributes in the order that the user specified. There may be a permutation of of $vs(p_2)$ more suitable than $vs(p_2)$. In fact, a permutation $\langle X.a, X.c, X.b, \downarrow X, \downarrow Y\rangle$ serves this purpose. In the following section, we discuss a systematic way to find this permutation, and prove finding such

a permutation is always possible and correct in the sense that the permuted visit sequence can evaluate the attributes as specified by the original attribution rules.

4. Algorithm and Proof

As described in the former sections, we prefer using normalized dependencies to avoid bogus cycles; we use $NDDP(p)$ instead of $DDP(p)$ in step 1-1 of the construction. However the resulting visit sequences might conflict with the original attribution rules. The former example showed us that the permuted visit sequences could help this situation. Assuming we can always permute the visit sequences, the new construction steps become as follows.

1. For each $p : X_0 \rightarrow X_1 \cdots X_{np} \in P$, we construct $IDP(p)$ by the following algorithm.
   1-1. First let $IDP(p) = NDDP(p)$
   1-2. Let $IDP(p) = IDP(p)^+$
   1-3. Let $IDP(p) = IDP(p) \cup \{(X_i.a, X_i.b) | \exists_{q:Y_0 \rightarrow Y_1 \cdots Y_{nq}} X_i = Y_j \wedge (Y_j.a, Y_j.b) \in IDP(q)\}$
   1-4. Repeat step 1-2 and 1-3 until no $IDP(p)$ changes.
2. Condition: each $IDP(p)$ must be acyclic.
3. For each $X \in T \cup N$, construct $IDS(X) = \{(X.a, X.b) | \exists_{p:X_0 \rightarrow X_1 \cdots X_{np}} X = X_i (0 \leq i \leq np) \wedge (X_i.a, X_i.b) \in IDP(p)\}$
4. For each $X \in T \cup N$, construct a partition $A_1(X), A_2(X), \ldots, A_{mx}(X)$ of attributes $A(X)$ from the dependencies in $IDS(X)$ such that $(X.a, X.b) \in IDS(X)$ implies $X.a \in A_i(X) \wedge X.b \in A_j(X) \wedge i \leq j$.
5. For each $p : X_0 \rightarrow X_1 \cdots X_{np}$, construct a visit sequence $vs(p)$ from partitions of $A(X_i) (0 \leq i \leq np)$, and dependencies in $IDP(p)$.
6. For each p, if $DDP(p)$ conflicts with the resulting visit sequence $vs(p)$, find a permutation of $vs(p)$ which does not conflict with $DDP(p)$ and all other attributes are successfully evaluable as before.

In the rest of this section, we present a simple algorithm to find the permutations described in step 6 above, and prove the algorithm is correct. First of all, by comparing the definitions of $DDP(p)$ and $NDDP(p)$, we notice the only dependencies included $DDP(p)$, but missing in $NDDP(p)$ are the dependencies between two attributes which are both defined in the context of p. Let $E(p)$ be such dependencies, namely $E(p) = \{(X_u.a, X_v.b) | X_u.a, X_v.b \in AF(p)\}$. Since $vs(p)$ is created from $NDDP(p)$, it reflects all dependencies of $DDP(p)$ but $E(p)$. The conflicting problem in step 6 arises if and only if $E(p)$ conflicts with $vs(p)$. More precisely for each p, the constructed visit sequence $vs(p)$ form step 1 to 5 is unusable if $E(p) \cup \{(X_i.a, X_j.b) | X_i.a$ appears before $X_j.b$ in $vs(p)\}$ is cyclic. The algorithm to make this acyclic by permuting $vs(p)$ is shown as follows.

Algorithm Since $E(p)$ is acyclic, $E(p)$ can be seen as a partial order on the attributes in $E(p)$. First sort these attributes topologically, and assign a number for each of them such that if there is a dependency $(X_u.a, X_v.b)$ in $E(p)$, $X_u.a$ has a number less than that of $X_v.b$.

Let $n(X.a)$ denote the number assigned for $X.a$. Then a binary relation $<$(less than) over the elements of $E(p)$ can be defined as:

$$< = \{((x,y),(x',y')) \mid (x,y),(x',y') \in E(p), n(y) < n(y') \vee (n(y) = n(y') \wedge n(x) < n(x'))\}$$

This tells us when we compare two dependencies, first compare by the second components, then compare the first one to break ties. Using $<$, we can construct a queue of dependencies by sorting the elements of $E(p)$. Let $Q = q_1, q_2, \ldots, q_{|E(p)|}$ be a queue of the dependencies of $E(p)$ such that $q_1 < q_2 < \cdots < q_{|E(p)|}$. For each $q = (X_u.a, X_v.b)$ in Q, there must be two distinct elements s_i and s_j in $vs(p)$ such that $s_i = X_u.a$, and $s_j = X_v.b$. Remember that $AF(p) = \{X_u.a | X_u.a \leftarrow f(\cdots) \in R(p)\}$. Thus everything defined in the context of p must appear in $vs(p)$ as a definition of some attribute. The algorithm to patch $vs(p)$ is:

Repeat until Q becomes empty:

1. Let $q = (X_u.a, X_v.b)$ be the last element of Q, and delete q from Q.
2. Let $vs(p) = \langle s_1, s_2, ..., s_m \rangle$.
3. Find s_i and s_j in $vs(p)$ such that $s_i = X_v.b$ and $s_j = X_u.a$.
4. If $i < j$, move s_j in front of s_i in $vs(p)$.

Proof First, we will show each iteration keeps the visit sequence $vs(p)$ consistent with $NDDP$. Notice that s_j is a definition of some attribute. Changing the position of s_j in step 4 does not affect the order of the visits to other nodes. This follows that we only have to prove every element $s_k (1 \leq k \leq m)$ of $vs(p)$ is OK if s_j is moved in step 4 including s_j itself. Here OK means:

1. If s_k is $X.a$, all attributes needed to define $X.a$ are available before the visit reaches s_k in $vs(p)$.
2. If s_k is $\downarrow_w X_u$, all attributes that are defined in the context of p, and used to define the attributes in the wth visit of X_u must be defined before the visit reaches s_k in $vs(p)$.

If s_j is not moved in step 4, $vs(p)$ will not change. Thus we have to make sure that all elements of $vs(p)$ are OK after each iteration of the algorithm only when s_j is moved. The following three cases must be considered.

1. If $k < j$, s_k is OK since s_k is not dependent on s_j anyway.
2. If $k = j$, for s_j itself, we have to prove that enough attributes have been defined already to define s_j. This is obvious since when the $NDDP(p)$ was computed, we made s_i dependent on all attributes on which s_j was dependent originally by taking the transitive closure of dependencies. Hence all attributes needed for s_i is also sufficient for s_j.
3. If $k > j$, s_k is OK since all attributes necessary for s_k are still available for s_k as described above. The only difference is that s_j is defined earlier than before so that the lifetime of $X_u.a$ becomes longer.

We showed that at the end of each iteration, all elements of $vs(p)$ are OK. i.e. there are no inconsistent dependencies in $vs(p)$ after each iteration. Hence when the iteration terminates, the order of elements in $vs(p)$ must be consistent with $NDDP$.

Second, we will show that the resulting $vs(p)$ is consistent with the dependencies in $E(p)$ by induction. More formally $vs(p) = \langle s_1, s_2, \ldots, s_m \rangle$ is consistent with $E(p)$ if $E(p) \cup \{(s_x, s_y) | x < y \wedge s_x, s_y$ are in $vs(p)\}$ is acyclic. Let $E_i(p)$ be the set of dependencies already integrated with $vs(p)$ at the end of the ith iteration.

1. When $i = 0$, $E_0(p)$ is empty. Hence $vs(p)$ is consistent with $E_0(p)$.
2. When $i = k (k > 0)$, assume $vs(p)$ is consistent with $E_k(p)$.
3. When $i = k + 1$, there are two cases.

   If we did not move the position of s_j in step 4 of the current iteration, $vs(p)$ does not change. Thus $vs(p)$ is consistent with $E_{k+1}(p)$ by assumption 2 above.

   Suppose $q = (s_z, s_x)$, and s_z was moved in front of s_x. Let $vs'(p)$ be the visit sequence before this change. If this move made $vs(p)$ inconsistent with $E_{k+1}(p)$, then there must be s_y in $vs'(p)$ such that $vs'(p) = \langle s_1, \ldots, s_x, \ldots, s_y, \ldots, s_z, \ldots, s_m \rangle$, and $(s_y, s_z) \in E_k(p)$. Here the existence of $(s_z, s_x) \in E(p)$ implies $n(s_z) < n(s_x)$, and further more $(s_y, s_z) < (s_z, s_x)$. Since all elements of Q are sorted, q must be less than every element in $E_k(p)$. But this contradicts against what we have, namely $(s_y, s_z) < q = (s_z, s_x)$ and $(s_y, s_z) \in E_k(p)$. Because of this contradiction, this move must be consistent with $E_{k+1}(p)$ always. After the last iteration, $vs(p)$ is consistent with $E(p)$, and this is the desired result.

Fig. 5

$A(IfStmt) = \{IfStmt.else,\ IfStmt.exit\}$
$A(Exp) = \{Exp.type\}$
$A(Id) = \{Id.symtab\_entry\}$

$p_1 : Stmt \rightarrow IfStmt$

$p_2 : Stmt \rightarrow AssignStmt$

$p_3 : IfStmt \rightarrow$ *'if'* Exp *'then'* $Stmt_1$ *'else'* $Stmt_2$
$R(p_3) = \{IfStmt.else := \text{gen\_label}(), IfStmt.exit := \text{gen\_label}(), Exp.type := bool\}$

$p_4 : AssignStmt \rightarrow Id$ *':='* Exp
$R(p_4) = \{Exp.type :=$ get_type$(Id.symtab_entry)\}$

---

5. Local Attributes

It is often convenient to have attributes attached to productions as well as to symbols of AGs. Fig. 5 shows part of an AG for a small language consisting of only assignments and if statements. Assume $IfStmt.else$ and $IfStmt.exit$ are attributes that we intend to store the values of labels for code generation. The attribution rules for these attributes are simplified, yet typical ones for the conditional constructs of usual programming languages. However this AG is somewhat unnatural in the following sense.

1. The chain rules $Stmt \rightarrow IfStmt$, and $Stmt \rightarrow AssignStmt$ are introduced only to associate attributes $IfStmt.else$ and $IfStmt.exit$ to the nonterminal $IfStmt$. i.e. we had to add two extra productions and two nonterminals just to find the place where the attributes can be attached.
2. These attributes are only referenced inside of p_3; no attribution rules but those of p_3 reference them.

This observation implies we sometimes need to modify the underlying CFG for the attributes even if they are very local to some specific attribution rules in the CFG. If we were allowed to have attributes not only for symbols, but also for productions of the CFG, we would not have this situation. Here we introduce a concept of *local attributes.*

Definition 1. An attribute a attached to a production p is denoted by $p.a$, and its scope is within all attribution rules associated with p. The attribute $p.a$ is called a *local attribute* of p, or it is *local* to p. The set of all attributes *local* to p is denoted by $A(p)$.

We can apply and define the local attributes just like other attributes in the attribution rules. Again the only difference are their scopes; they must be defined by some attribution rule of the production to which they are attached, and can be referenced by any attribution rules for the production. In this framework the attribution rules of a production $p : X_0 \rightarrow X_1 \cdots X_{np}$ consists of attribution rules of the form $X_i.a := f(\cdots)$ as well as $p.b := g(\cdots)$. The dependencies among local attributes and other attributes are also defined.

Fig. 6

$$
\begin{aligned}
&A(p_1) = \{p_1.else,\ p_1.exit\} \\
&A(Exp) = \{Exp.type\} \\
&A(Id) = \{Id.symtab\_entry\} \\
\\
&p_1 : Stmt_1 \rightarrow \text{'}if\text{'}\ Exp\ \text{'}then\text{'}\ Stmt_2\ \text{'}else\text{'}\ Stmt_3 \\
&R(p_1) = \{p_1.else := \text{gen\_label}(), p_1.exit := \text{gen\_label}(), Exp.type := bool\} \\
\\
&p_2 : Stmt \rightarrow Id\ \text{':='}\ Exp \\
&R(p_2) = \{Exp.type := \text{get\_type}(Id.symtab\_entry)\}
\end{aligned}
$$

Definition 2. The set of all dependencies involving local attributes of a production p is given by $LDDP(p) = \{(a,b)|a \in A(p) \vee b \in A(p)\}$.

With the notion of *local attributes*, we can rewrite the previous AG as shown in Fig. 6. Note the artificial chain rules and extra nonterminals went away. The attributes $IfStmt.else$ and $IfStmt.exit$ became $p_1.else$ and $p_1.exit$ respectively, and they are local to p_1.

The construction of the visit sequences based on OAG needs few modifications to accommodate the local attributes because the scope of local attributes are limited to attribution rules of each production. This means the local attributes do not affect the interfaces between the visit sequences in a direct way, and all the dependencies that involve local attributes can be removed as we have removed the dependencies between attributes which were both defined in the context of the production by using $NDDP$s instead of DDPs. After a visit sequence has been constructed, we can insert the definitions of local attributes into the visit sequence, and obtain the final visit sequence that evaluates both attributes of the symbols and the production. The new and the final construction steps of the visit sequences are shown below.

1. For each $p : X_0 \rightarrow X_1 \cdots X_{np} \in P$, we construct $IDP(p)$ by the following algorithm.
   1-1. First let $IDP(p) = (DDP(p) \cup LDDP(p))^+ - \{(a,b)|a,b \in AF(p) \vee a \in A(p) \vee b \in A(p)\}$
   1-2. Let $IDP(p) = IDP(p)^+$
   1-3. Let $IDP(p) = IDP(p) \cup \{(X_i.a, X_i.b)|\exists_{q:Y_0 \rightarrow Y_1 \cdots Y_{nq}} X_i = Y_j \wedge (Y_j.a, Y_j.b) \in IDP(q)\}$
   1-4. Repeat step 1-2 and 1-3 until no $IDP(p)$ changes.
2. Condition: each $IDP(p)$ must be acyclic.
3. For each $X \in T \cup N$, construct $IDS(X) = \{(X.a, X.b)|\exists_{p:X_0 \rightarrow X_1 \cdots X_{np}} X = X_i (0 \leq i \leq np) \wedge (X_i.a, X_i.b) \in IDP(p)\}$
4. For each $X \in T \cup N$, construct a partition $A_1(X), A_2(X), \ldots, A_{mx}(X)$ of attributes $A(X)$ from the dependencies in $IDS(X)$ such that $(X.a, X.b) \in IDS(X)$ implies $X.a \in A_i(X) \wedge X.b \in A_j(X) \wedge i \leq j$.
5. For each $p : X_0 \rightarrow X_1 \cdots X_{np}$, construct a visit sequence $vs(p)$ from partitions of $A(X_i) (0 \leq i \leq np)$, and dependencies in $IDP(p)$.
6. For each $vs(p) = \langle s_1, s_2, \ldots, s_{mp} \rangle$, attach every element of $A(p)$ to its tail in any order, and construct a new visit sequence $vs'(p) = \langle s_1, s_2, \ldots, s_{mp}, l_1, l_2, \ldots, l_{|A(p)|} \rangle$, where $l_i (1 \leq i \leq |A(p)|) \in A(p)$.
7. For each p, if $DDP(p) \cup LDDP(p)$ conflicts with the new visit sequence $vs'(p)$, find a permutation of $vs'(p)$ which does not conflict with $DDP(p)$ and all other attributes are successfully evaluable as before. In the previous construction, the permutation was based on $E(p)$, but we use $E(p) \cup LDDP(p)$ instead of $E(p)$.

In this construction, we look at $E(p) \cup LDDP(p)$ as the partial order, and create the relation $<$ between dependencies as we did before based on $E(p) \cup LDDP(p)$.

6. Conclusion

We have seen the concept of local attributes and how they can be integrated into OAG-based, visit-oriented attribute evaluators. As mentioned earlier, each visit sequence can be seen as a co-routine. In this view, the conventional attributes associated with symbols are parameters and forms the interface of the co-routine. As we need local variables, it is convenient to have local attributes. They hold useful values to compute other attributes so that we can avoid writing same expressions many times in the attribution rules. As local variables helps us to write more readable and understandable routines, the concept of local attributes allows us more natural and readable attribution rules.

ACKNOWLEDGMENTS. I would like to thank Dr. Waite for his kind advises and suggestions for the usefulness of the concept of local attributes.

REFERENCES

1. Knuth, D. E. 'Semantics of context-free languages', Math. Syst. Theory 5, 1968, 127-145.
2. Kastens, U., Hutt, B., and Zimmermann, E. 'GAG: A practical compiler generator', Lecture Notes in Computer Science, vol. 141. Springer-Verlag, New York, 1982.
3. Kastens, Uwe 'LIGA/LIDO: A Specification Language for Attribute Grammars, University at Paderborn, Oct. 1989.
4. Farrow, R., 'LINGUIST-86 Yet Another Translator Writing System Based On Attribute Grammars', Special Interest Group on Programming Languages 17, 1982.
5. Koskimies, K., Räihä, K., and Sarjakoski, M., 'COMPILER CONSTRUCTION USING ATTRIBUTE GRAMMARS', Special Interest Group on Programming Languages 17, 1982.
6. Kastens, U., 'Ordered attributed grammars', *Acta Inf. 13*, 1980, 229-256.
7. Gray, R., Heuring, V., Krane, S., Sloane, A., and Waite, M., 'Eli: A Complete Compiler Construction System', SEG 89-1-1, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO, June 1989.
8. Bochmann, G. V., 'Semantics Evaluation from Left to Right', *Commun. ACM 19*, 2, Feb. 1976, 55-62.
9. Engelfriet, J., and Filé, G., 'Passes, Sweeps, and Visits in Attribute Grammars', *Journal of ACM*, 4, Oct. 1989, 841-869.
10. Kastens, U., 'Lifetime Analysis for Attributes', *Acta Inf. 24*, 1987, 633-651.
11. Waite, W., and Goos, G, 'COMPILER CONSTRUCTION', Springer-Verlag, New York, 1984.